# A Report on Inter-Process Communication (IPC) in Support of Landsat 7 LPS Interface Design Document

**Cliff Liu**

**Apr. 4, 1995**

## Abstract:

IPC (Inter-Process Communication) refers to the methods of sending data from one process to another asynchronously (the synchronous data passing for process and function invocation is via arguments). IPC mechanisms generally include files, pipes, FIFOs, message queues, shared memory, semaphores, and signals. Recent evolution of light-weight processes (a.k.a. threads) which use shared global data between the related processes provide more efficient inter-process communication. An example of that is SGI's ***sproc***ed processes in multi-threads parallel programming.

## Introduction:

This report concentrates on the general IPC mechanisms which are supported by most Unix systems (including SGI). These IPCs are commonly referred to as the traditional Unix IPCs, System V IPCs and BSD sockets. Being an application program interface (API) to the network protocols such as TCP, the BSD Socket is normally used for network communications between computers. Please note that even though database with SQL query interface has been determined by LPS as one of the IPC mechanisms, its applications should be confined to the less time-critical tasks such as report generation due to its slow speed. The description of database interface is not included in this report. Also note that semaphores and signals are generally used to inform another process of a condition instead of passing data. They are normally used with other IPC mechanisms such as the shared memory or message queues.

The characteristics and system calls involved for each IPC mechanism are listed below, followed by performance comparisons, portability issues and conclusions. The goal of this report is to provide readers a general knowledge of IPC mechanisms and as a reference in selecting the most efficient IPC method. But keep in mind that the IPC method of choice may be affected by the software architecture and design considerations.

**Traditional Unix IPCs:**

* Files
- A process writes to a file which will be read by another process.
- The processes do not need to be related.
- It does not require IPC system calls, most portable.
- Has potential concurrent access problem - one process reads from a file while another is writing to it. Use file locking or record (range) locking to prevent this
- It is a slower IPC but may be necessary for applications that involve large data files.
- Use open(), lseek(), read(), write(), lockf() system calls.

* Pipes
- Pipe is a one-way data string I/O: the output of one process is sent directly to the input of the other process. Use two pipes when two-way data flow is desired.
- Used for related processes such as parent and child or children of the same parent.
- Read and write are not guaranteed to be atomic, they can be interrupted and have the buffer intermixed with data from multiple readers or writers. This makes pipes unsuitable for more than two processes.
- Use pipe(), read(), write() system calls.

* FIFOs (AKA. Named pipe)
- Similar to a pipe, FIFO is also a one-way data string I/O. however, it has a name associated with it to allow unrelated processes to access it.

- Atomicity is guaranteed but this makes FIFOs slower than pipes (it may not to be true if a process writes more data then a pipe can hold).
- Use mknode(), open(), read(), write(), close() system calls.

* Signals
- A signal is a condition that can be sent to a process by the kernel or another process. A process provides signal handler to perform an action or allows default action to be taken. It can also mask out signals (other than SIGKILL and SIGSTOP).
- Signal is asynchronous, a program can receive a signal between any two instructions.
- A process can send signals to processes in the same group, with the same effective user ID, in another group. A superuser can send signals to all processes. No shared common kernel data structure is needed for a signal.
- signal() and kill() are ANSI C signal system calls. POSIX also uses kill(), but it replaces signal() with a set of functions (sigaction, sigprocmask,etc.)

**System V IPCs:**

- System V IPCs Include Message queues, Semaphores, shared memory
- The resource used by message queues, semaphore and shared memory will remain as a kernel data structure even after the process that created the resource terminates.
- All System V IPCs are system wide, any process can access an IPC channel identified by an integer ID.
- Use ipcs command to check the System V IPC configuration - the information stored in the kernel for each established IPC channel.

* Message queues
- Messages queues allow processes to send a structure (instead of streams used by pipes and FIFOs) to other unrelated processes.
- A process can either wait for a message or use a signal.

- All message queues are stored in the kernel, and have an associated message queue identifier (msgid).
- Many processes can send a message to the same message queue. The message type can be used to indicate which of the several different processes is the originator of a message (or the message priority for LPS' Status Messages). The receiving process can take off the first message of a selected message type.
- Use msgget () to create a new message queue or access an existing one.
- Use msgsnd() and msgrcv() to send and receive a message.
- Use msgctl() to remove or modify a message queue data structure from the kernel.

* Semaphores

- A semaphore is used for resource synchronization. In its simplest form, i.e. a binary semaphore of value 0 or 1 is used to protect a resource from being accessed by another process until the process currently using that resource releases it. A more general semaphore is a resource counter that indicates the number of available resources.
- System V implements semaphores in the kernel to guarantee atomicity.
- Use semget() to create or access an existing semaphore.
- Use semctl() to initialize or remove a semaphore.
- Use semop() to allocate or release resources controlled by a semaphore.

* Shared memory

- Shared memory is faster than any other method because it requires no data transfer. For pipes, FIFOs or message queues, data is copied from the sender's buffer into the kernel, and then copied from the kernel to the reader's buffer.
- It allows any number of processes, whether related or not, to access the same data segment as part of the virtual address space of each process.
- The steps in using shared memory:
    1. Use shmget() to create a shared memory in the kernel.

2. Determine a safe place in each process to attach the shared memory region.
3. Use shmat() to attach the shared memory to a process.
4. Use semaphores to synchronize concurrent access to the shared memory.
5. Detach the shared memory region.

**BSD sockets:**

- It provides application interface to network protocols (Unix domain or Internet domain - TCP/IP). The data transfer can be connection-oriented or connectionless.
- Two processes wishing to communicate set up a pair of sockets to create a communication channel between them. It allows communication between processes running on the same computer (Unix domain protocol) or on different computers on a network (Internet domain protocol).
- BSD implements pipes using sockets to connection-oriented Unix domain protocol (That's one reason why pipes may be slower due to the network overhead).
- Both computers on the network must have socket facility for this method to work.
- Asynchronous I/O can be used to have the kernel inform the process via a SIGIO signal when data is arrived at a socket.
- Following is a client-server example using sockets and the connection-oriented protocol :

  - Server implementation:

    1. Call socket() to create a socket with a protocol type. It returns a socket descriptor
    2. Initialize a pointer and the content of the sockaddr structure. (AF_UNIX - two processes on the same computer)
    3. Assign a name to socket and register server's address - bind().
    4. Listen for connection requests from clients - listen().

5. Wait for connection request from client, receive client's address and create a new socket for the client - accept().
6. Fork a child to serve the client if it is a concurrent server, otherwise serve the client itself.
7. Close the socket when service is complete - close().

- Client implementation:

1. Same as 1&2 above
2. Establish connection with server via connect() and register client's address.
3. Use read() and write(), or use send(), sendto(), recv(), recvfrom().

**Portability:**

Despite using the fairly standard System V IPC methods described above, changes to the IPC programs may still be necessary when porting an application to another system because of the difference in the kernel data structure and the IPC tunable parameters. For example, SGI supports up to 32K bytes per message which is normally 8K in other Unix systems. The portability issues for inter-process communication will be fully defined in the POSIX.4 Real-Time system standard.

In addition, some of the system calls described in this report are not POSIX.1 compliant, for example the ANSI C signal() is replaced by sigaction() in POSIX.1, but both of them are currently portable.

It is obvious that use of system specific IPC methods such as SGI's Arena shared memory and multithreads programming will require significant code changes for porting.

**Performance:**

This section presents the results of two IPC performance studies conducted by Ron Leach in the "Advanced Topics in UNIX" and by CSC PACOR II team on the RTOS software.

Ron Leach compared efficiency of the various IPC mechanisms on three different platforms. He found out that shared memory was the fastest IPC among all three systems when used without synchronization control. But most concurrent processes require some sort of synchronization to protect data integrity. The shared memory with semaphore performed slower than pipes, FIFOs and message queues, especially when a large amount of data (1000 byte) is passed. This was probably caused by a context switch due to the time for ***mencopy()*** to copy 1000 bytes was longer than the allocated time slice for the process. An alternate code design and a careful scheduling should be able to avoid a context switch.

Also common to all systems is that files always performed slowest. Following is a brief summary of the top IPC performers on each system.

> AT&T 3B2/310 (SVR3) - To transfer a small amount of data ( 10 - 100 bytes) , message queue was the fastest, followed by FIFOs, pipes. The difference in performance between pipes and FIFOs was small. However, when transferring 1000 bytes of data, pipes and FIFOs outperformed message queue.
>
> Sun 3/60 (SunOS 3.5)- Again message queues outperformed other IPC methods, and it remains true even with larger data. FIFOs were faster than pipes on Sun.
>
> Sun SPARC 2 (Solaris 1.1) - Same orders as above, i.e. message queue, FIFOs and pipes. Followed closely by the shared memory with semaphores which performed better than the other two systems.

CSC PACOR II team conducted an inter-process communication performance study of the IPC mechanisms such as FIFOs, BSD sockets and message queues to communicate between RTOS tasks during the software critical design phase. The tests were not performed in a highly controlled environment as its goal was to get a quick comparison of the IPC capabilities. Note that the message queue test was not run for data greater than 1K bytes for both tests due to size limitation (SGI's message size can be up to 32K bytes).

The tests were performed on two different Sun SPARC workstations running two versions of the Sun OS - 4.1.2 and 5.1. The results of the tests on Sun OS 4.1.2 showed that both message queues and FIFOs had better throughput than sockets. The tests running on Sun OS 5.1 including TLI showed a similar result that message queue came first (up to 1K data size), followed by sockets and TLI, then the FIFOs.

## Conclusions:

The performance of IPC mechanisms can vary with different platforms and operating systems. There is no known performance study performed on the SGI IRIX. But it is safe to expect better performance from the newer Unix systems in which IPC mechanisms are fine-tuned.

Based on the above performance studies and the characteristics of each IPC mechanism, the SVR4 shared memory with synchronization control and the message queues seem to be the best choice as the IPC methods between LPS subsystems. As for the interface between LPS and LP DAAC, the BSD sockets and FTP (not described in the report) may be used. This interface depends on the forthcoming ICD between LPS and LP DAAC.

The communication between LPS subsystems will be either one-writer, one-reader (eg. RDPS-MFPS) or one-writer, many-reader (eg. MFPS-PCDS/IDPS) fashion. Synchronization of the shared memory for the former can be achieved using a simple data flag to avoid the overhead of kernel processing, or using semaphores for the latter. Besides better performance, other advantages of message queues over pipes and FIFOs are that they support structure passing instead of passing byte strings and that the message type can be used as a subsystem ID or message priority. The LPS Status Messages and some MACS directives are good candidates for message queues.

When time critical code demands the use of SGI's light-weight processes to run multiple threads in parallel, the shared data will be the most efficient IPC method.

A final note, the OSF Distributed Computing Environment (DCE) to be used by the Renaissance project as IPC mechanism is not available for SGI platform. The remote procedure calls (RPC) however may be considered as an alternative interface between LPS and LP DAAC.